Pattern Based Processing of XPath Queries*

Gerard Marks
Interoperable Systems Group
Dublin City University
Dublin, Ireland
gmarks@computing.dcu.ie

Mark Roantree
Interoperable Systems Group
Dublin City University
Dublin, Ireland
mark.roantree@computing.dcu.ie

ABSTRACT

As the popularity of areas including document storage and distributed systems continues to grow, the demand for high performance XML databases is increasingly evident. This has led to a number of research efforts aimed at exploiting the maturity of relational database systems in order to increase XML query performance. In our approach, we use an index structure based on a metamodel for XML databases combined with relational database technology to facilitate fast access to XML document elements. The query process involves transforming XPath expressions to SQL which can be executed over our optimised query engine. As there are many different types of XPath queries, varying processing logic may be applied to boost performance not only to individual XPath axes, but across multiple axes simultaneously. This paper describes a pattern based approach to XPath query processing, which permits the execution of a group of XPath location steps in parallel.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems Query processing

General Terms

Algorithms, Performance, Languages

Keywords

XML Storage, Mapping techniques, XPath Optimisation, Patterns

1. INTRODUCTION

In many cases, document databases are the choice for organisations such as government agencies over more traditional data-centered databases. As a result, there are now numerous XML database implementations [9, 10, 11]. For this purpose, the W3C [12] have standardised languages such as XPath and XQuery with path expressions to process the tree structures of these databases.

A problem with XML databases is associated with poor performance for certain database queries. While we have addressed this issue in our FASTX XML database using an optimised index [13, 17] and a metamodel for XML databases [16], we now optimise further by extracting metadata from the query itself. Using this query metadata, we can group a number of XPath location steps that have similar characteristics together, and process them in parallel.

1.1 Background and Contribution

Given an XPath query, the processor must locate the result set that satisfies the content and structural conditions specified by the query. Suitable indexing structures together with query processing strategies can significantly improve the performance of this matching operation. There are a number of index-based query processing strategies utilised by XML databases [1, 2, 17, 16, 22, 23].

In this paper, we discuss our approach to XML query optimisation. Our approach is to parse the XPath query in order to separate and classify a number of location steps based on:

1. Their axis type.
2. The presence of predicates.
3. The location of predicates within the query.

Using these three factors we implemented a number of reusable solutions (*patterns*), each of which parse a group of XPath location steps, and transform them to SQL for execution across our relational XML repository. This allows us to use the most efficient processing logic to process each query segment in order to boost the performance of the overall query. The purpose of this paper is to demonstrate a proof-of-concept for this approach by describing those *patterns* which currently map a group of XPath location steps to SQL and how using a subset of XPath queries, we can demonstrate good performance gains. A more complete XPath to SQL mapping process forms part of our ongoing work.

Contribution. This work presents a *pattern* based approach for optimising XPath queries by transforming a group of XPath location steps into SQL commands that can be executed in parallel. We exploit our index structure to process XPath location steps in parallel when one or more of the patterns described in section 3 may be applied to the query

*Funded by Enterprise Ireland Grant No. CFTD/07/201

IDEAS08 2008, September 10-12, Coimbra [Portugal]
Editor: Bipin C. DESAI

while the remaining (individual) location steps are processed using the techniques described in existing work [16].

Much of the related research in this area boosts the performance of XPath queries by targeting each of the 13 XPath axes individually where the resulting context nodes from each step serve as input to the next location step in order [28, 17, 16].

An extension of this idea is to process multiple location steps in parallel [19]. We extend this approach by identifying a number of patterns that permit parallel processing, and use query metadata to correctly apply the relevant patterns to the query.

Paper Structure In 2, we describe XPath processing and the serial nature of location step evaluation before introducing our framework for parallel evaluation of location steps. In 3, we describe the patterns identified for parallel processing of XPath location steps. In 4, we describe our experiment sets executed across two different databases and provide an analysis of the outcome. In 5, we discuss related research in the area of XML query optimisation. Finally, in 6, we provide some conclusions.

2. FRAMEWORK ARCHITECTURE

2.1 Overview

An XPath query is a "location path" to a set of nodes in an XML document. Each component of this path is a "location step" made up of a *node test*, *axis* and zero or more *predicates*. The XPath query language has 13 axes (eg. *descendant*, *following*, *ancestor*) and allows an XML document to be queried as a tree of nodes which returns subtrees relevant to the XPath query.

In order to process a number of location steps in parallel we must identify candidate location steps within the query and group them. We call this group of location steps a *partial query*.

A breakdown of an XPath query into its individual location steps is shown in table 1.

EXAMPLE 1. *Find all payments made in the North American region using a Credit Card when only one item was purchased.*

/site/regions//item[location = 'United States'][quantity = '1'][.//payment = 'Creditcard']/parent::namerica

Step	Axis	NodeTest	Predicates
1	child	site	
2	child	regions	
3	descendant	item	**child::location = 'United States'** child::quantity = '1' **descendant::payment = 'Creditcard'**
4	parent	namerica	

Table 1: Location Steps for *Example* 1

A system which processes XPath serially will target each location step individually and from left to right the output of one step serves as the input to the next. The output of the right most step is the final result for the query.

In a parallel system, a number of location steps may be processed simultaneously using a single SQL statement, avoiding unnecessary application layer processing for each location step. Our approach provides an optimisation framework in which several patterns are exploited to improve query performance.

2.2 Optimisation Framework

By analysing an XPath query, and extracting metadata, we have found that much of this information is repeated throughout an arbitrary group of queries. Thus, we have identified a number of patterns which enable us to reuse identified solutions for queries which fall under a particular classification. A detailed description of patterns is presented in section 3.

The generic steps in our process for all patterns involves parsing a query to generate the necessary metadata to permit a classification, selection and ordering process in steps 1 and 2. In Step 3, the relevant algorithm(s) are invoked to transform the XPath query to SQL prior to query execution.

1. **Classification Process.** This step takes the XPath query as input and parses each location step in order extracting metadata. The breakdown of the example in Table 1 shows that the query has four location steps containing the: {child, descendant and parent} axes, and the third location step has three predicates. The following properties are used to define the classification rules for each pattern:

   PROPERTY 1. *The Axes to be processed in parallel must belong to the following set:* {***descendant, descendant-or-self, child, self and attribute***}.

   PROPERTY 2. *Only the right most location step may contain predicate filters.*

   Later, in section 3, we describe four patterns for XPath processing: the FullPath, LeafPath, Filter, and Parent patterns. At this point, we provide an overview of how they are selected. Using the query metadata extracted (see table 1) the classification rules are as follows:

   - The FullPath Pattern may be applied to a group of consecutive location steps when properties 1 and 2 hold.
   - The LeafPath Pattern has the same classification rules as the FullPath Pattern and thus, both are always an option when properties 1 and 2 hold. A decision as to the highest performing paterrn is made at the next step.
   - The Filter Pattern may be applied to a group of consecutive location steps when property 1 holds.
   - The Parent Pattern may be applied when a location step contains the Parent axis and is immediately preceded by the FullPath or LeafPath patterns. It may also be applied where parallelism does not exist as a single location step can be integrated into the Parent Pattern's processing logic.

   The output from this classification process is the pattern(s) which may be applied, and the location steps they apply to. In the case of Example 1, the output from this step is:

(a) The FullPath pattern may be applied to location steps 1, 2 & 3.

(b) The LeafPath pattern may be applied to location steps 1, 2 & 3.

(c) The Parent pattern may be applied to location step 4 and thus, allows a two-pattern combination using one of the patterns above.

As multiple patterns are identified for various location steps, the next step is to select the highest performing of the patterns identified above.

2. **Pattern Selection.** The selection process relies on the metadata we can extract from the query *and* the XML repository. This metadata falls into three categories:

   (a) Data related to the semantics of the XPath query language.

   (b) Data related to the structure of the query.

   (c) Data related elements within the XML repository.

   We present examples of this selection process in the experiments section 4. In all cases, experimental evaluation has shown, it is optimal to select the pattern which maximises parallelism. Taking the inputs from the previous step the Parent Pattern in conjunction with the FullPath or LeafPath Pattern can process all four location steps. Thus, either the Parent-FullPath or Parent-LeafPath combination of patterns will be selected.

   To select between these two combinations the processor will find the number of FullPath instances using metadata from the XML repository. Experimental evaluation indicates that FullPath Pattern performs better when the number of FullPath instances is less than 20,000. The number of FullPath instances for example 1 is 43,500 therefore, Parent-LeafPath combination is chosen. A detailed description of this selection process is provided in section 4.

   Now that we have identified the patterns to be applied, the next step is the XPath query transformation to SQL.

3. **Query Transformation.** The input and output for this step are:

   (a) **Input:** The location steps to be processed, and the pattern to be applied.

   (b) **Output:** A single SQL statement.

   In the case of our example the input is all four location steps (see table 1) as the Parent-LeafPath combination may be applied to the entire query. The parent pattern is then executed (taking the LeafPath.SQLMAP() as input) to parse the query extracting the required parameters for execution of the Parent.SQLMAP(**Input:** LeafPath.SQLMAP) algorithm.

   Each pattern has an algorithm which translates the relevant group of XPath location steps to SQL. The mapping algorithm for each pattern is identified by: PatternName.SQLMAP(), these are shown in section 3.

   The output of Parent.SQLMAP() is a single SQL statement that serves as the input for the next step.

4. **Query Execution.** In the final step, SQL statements are executed to return a set of nodes, which are used to build the final result set (one or more XML documents).

3. PATTERNS

It has been suggested [21] that a pattern requires three properties:

1. **Context:** A situation giving rise to a problem.

2. **Problem:** The recurring problem arising in that context.

3. **Solution:** A proven resolution of the problem.

By analysing a large group of XPath queries across many databases, we identified a number of situations where the same solution could be applied to solve issues relating to query performance. In each case, the solution involves processing a group of location steps that have similar query structures and XPath language properties in parallel.

Currently there are four patterns in FASTX, each of which is described in this section. We use Null Pattern (NP) to denote where the individual location step optimiser is used, the details of which can be found in our previous work [19]. Table 2 shows how the patterns may be used.

Level	Combination
1	{FPP}, {LPP}, {NP}
2	{FPP, PP}, {FPP, FP} {LPP, PP}, {LPP, FP} {NP, PP}, {NP, FP}

Table 2: Pattern Usage Combinations

Only patterns at level 1 may be executed in isolation. The Parent Pattern (PP) is used in conjunction with NP, FullPath (FPP) or LeafPath (LP) pattern. The Filter Pattern (FP) may be used with FPP, LP, and NP.

Before we discuss the patterns in detail we must introduce a number of variables which are used throughout the algorithms in this section.

`BIT` is used to represent the Base Index Table in the relational index structure. Our index uses an extended PreOrder encoding scheme [17] to uniquely identify individual elements and attributes in the XML document. We use `PRE` to represent PreOrder values in the index. To permit hierarchal traversal of an XML document, the `PARENT` element in the `BIT` holds the PreOrder value of each element and attributes parent. `RFINAL` holds the final SQL statement resulting from the algorithm.

3.1 FullPath Pattern

A FullPath is a path from the document root node to the node test of the right most location step in a group. A LeafPath (described below) extends the FullPath to the target nodes of any predicates that exist. Using the query in example 1 table 3 shows this comparison.

The FullPath pattern is distinguished by the fact that properties 1 and 2 must hold, and the structure of the SQL

Table 3: Compare Paths

FPP Paths (T1)	LP Paths (T1)
/site/regions/africa/item	/site/regions/africa/item/location
/site/regions/asia/item	/site/regions/asia/item/location
/site/regions/australia/item	/site/regions/australia/item/location
/site/regions/europe/item	/site/regions/europe/item/location
/site/regions/namerica/item	/site/regions/namerica/item/location
/site/regions/samerica/item	/site/regions/samerica/item/location
	/site/regions/africa/item/quantity
	/site/regions/asia/item/quantity
	/site/regions/australia/item/quantity
	/site/regions/europe/item/quantity
	/site/regions/namerica/item/quantity
	/site/regions/samerica/item/quantity
	/site/regions/africa/item/payment
	/site/regions/asia/item/payment
	/site/regions/australia/item/payment
	/site/regions/europe/item/payment
	/site/regions/namerica/item/payment
	/site/regions/samerica/item/payment

statement resulting from the FullPath.SQLMAP() (see algorithm 1). Examples of this are shown in the experiments section 4.

Taking a group of location steps which adhere to these properties, the query processor identifies the parameters required for the FullPath.SQLMAP() algorithm.

3.1.1 Parameter Extraction

There are two outputs: a collection of FullPaths (T_1) see table 3, and a set of query tuples containing node identifiers and filters, (T_{2a} ,T_{Pa}) to (T_{2n},T_{Pn}). In the case of example 1 these are:
T_{2a}=location; T_{Pa}='United States';
T_{2b}=quantity; T_{Pb}='1'.
T_{2c}=payment; T_{Pc}='Creditcard'.

Algorithm 1 FullPath.SQLMAP()

```
String[] S;
String RFINAL;
Collection T1; // FullPaths
Collection T2; // Predicate expressions
Collection TP; // Predicate values
for each T2x and TPx do
  S[1] += "SELECT PARENT FROM BIT WHERE
  NAME = ' "+T2x+" ' ";
  if value filter exists then
    S[1] += " AND VALUE = ' "+TPx+" ' ";
  end if
  if node test is attribute then
    S[1] += " AND N_TYPE = ATTRIBUTE ";
  else
    S[1] += " AND N_TYPE = ELEMENT ";
  end if
  if more predicate expressions in T2 then
    S[1] += " INTERSECT ";
  end if
end for
for each T1x do
  S[2] += ' "+T1x+" ';
  if more paths then
    S[2] += " OR FULLPATH = ";
  end if
end for
if no predicates then
  RFINAL = "SELECT PRE FROM BIT WHERE
  FULLPATH = ' "+S[2]+" ' ";
else
  RFINAL = S[1] + " AND PARENT IN (SELECT PRE
  FROM BIT WHERE FULLPATH = ' "+S2+" ') ";
end if
```

3.2 LeafPath Pattern

The LeafPath pattern is so called because it extends the path from the document root node directly to the target node set of each predicate. The LeafPath pattern is a variation of the FullPath pattern and serves the same purpose (i.e. locations steps must adhere to properties 1 and 2). However, experimental evidence indicates (see section 4) that good performance gains result from selecting between these two patterns. The selection process requires one extra step for these two patterns in addition to query metadata.

We must find the number of instances of each FullPath from the index. The performance hit is low as the number of instances of each FullPath is calculated at index creation time. As stated above, the FullPath pattern performs better when the number of FullPath instances is in the range less than 20,000 PreOrder [17] nodes. After this point experimental analysis indicates that the LeafPath pattern performs better. The cause of this variation in performance is related to the SQL statements derived from each patterns SQLMAP(), we demonstrate a worked example of this in section 4.

3.2.1 Parameter Extraction

There are two outputs: a collection of LeafPaths (T_1), see table 3, and a set of query filters, (T_{2a} to T_{2n}). In the case of example 1 these are:
T_{2a}='United States';
T_{2b}='1'.
T_{2c}='Creditcard'.

3.3 Filter Pattern

This Filter pattern is used when property 1 holds, but property 2 does not. This means that the locations steps for this pattern may contain predicate filters which are not contained in the right most location step.

3.3.1 Parameter Extraction

There are two outputs: a SQL (S_1) statement resulting from the logic of the adjoining pattern (see combinations table 2), and a set of node tests (T_1) where the first element (T_{1a}) is the name of the node(s) required and the remaining node tests (T_{1b} to T_{1n}), let us filter backwards (using the parent field in the BIT) to allow only nodes that are permitted by the predicate filter to form part of the final result set.

3.3.2 Performance Cost

There will be an additional SQL substatement for each node test in T_1. This is a limitation as the overall processing time grows linearly by the number of elements in T_1. However, this pattern performs well when the size of T_1 is relatively small.

Algorithm 2 LeafPath.SQLMAP()

```
Collection T1; // collection of "FullPath" Strings
Collection T2; // collection of Predicate "VALUES"
String RFINAL = "(";
String NodeType;
for Each FullPath IN T1 do
  RFINAL += "SELECT "+NodeType+" FROM BIT
  WHERE FullPath = 'T1x'";
  if value filter exists then
    RFINAL += " AND VALUE = "+T2x+" ";
  end if
  if node test is attribute then
    RFINAL += " AND N_TYPE = ATTRIBUTE ";
  else
    RFINAL += " AND N_TYPE = ELEMENT ";
  end if
  if node test unchanged and more FullPaths then
    RFINAL += " UNION ";
  end if
  if node test changed then
    R += " ) INTERSECT ( ";
  end if
end for
RFINAL += ")";
```

Algorithm 3 Filter.SQLMAP()

```
String[] S;
String RFINAL ;
Collection T1; // Node Tests
S[2] += "SELECT PRE FROM BIT WHERE NAME =
' "+T1a+" ' ";
for each node test in (T1b to T1n) do
  S[3] += " AND PARENT IN (SELECT PRE FROM
  BIT WHERE NAME = ' "+T1x+" ' ";
end for
S[4] = " AND PARENT IN ("+S1+" ";
RFINAL = S[2] + S[3] + S[4] ;
for each node test in (T1b to T1n ) do
  RFINAL += "}";
end for
RFINAL += "}";
```

3.4 Parent Pattern

For an arbitrary query S, the Parent Pattern is used when:

1. A location step S_n contains the Parent axis.
2. The location step S_{n-1} is contained in the group of location steps processed by any of the patterns in the set: {FPP, LP, NP}.

The Parent.SQLMAP() involves retrieving the SQL output from the pattern used from the set in step 2 above, and joining it with the SQL for Parent pattern to form a single SQL statement.

3.4.1 Parameter Extraction

There are three outputs: a SQL ($S_{[1]}$) statement which is the output of the previous pattern executed to the immediate left this pattern, a node test N_T for the parent axis, and a set of query tuples containing node identifiers and filters, (T_{2a} ,T_{Pa}) to (T_{2n},T_{Pn}) where predicates exist for the parent axis.

Algorithm 4 Parent.SQLMAP()

```
String S1;
String NT;
String RFINAL;
Collection T2;
Collection TP;
S1= getSQL(pattern);
RFINAL += "SELECT PRE FROM BIT WHERE
NAME = ' "+NT+" ' AND PRE IN ( ";
if hasPredicates then
  for each predicate do
    RFINAL += " SELECT PARENT FROM BIT
    WHERE NAME = ' "+T2x+" ' ";
    if predicate has filter then
      RFINAL += " AND VALUE = ' "+TPx+" ' ";
    end if
    RFINAL += " INTERSECT ";
  end for
end if
RFINAL = "SELECT PARENT FROM BIT WHERE
PRE IN ("+S1+"))";
```

4. EXPERIMENTS AND RESULTS

4.1 Overview

In this section we evaluate sample queries and show the comparison times between FASTX and industry leader eXist to validate our approach. We also provide a work-through for selected queries to describe the pattern processing and evaluation in more detail. Experiments ran on a 2.66GHz Intel(R) Core(TM)2 Duo CPU machine with 3.25GB of RAM using Windows XP Professional operating system and the FASTX query processor was implemented using Java Virtual Machine version 1.6. We altered the default JVM settings from -Xmx128MB to -Xmx1024MB in order to permit queries which return large result sets. The eXist database (version=1.2.0-build=20080115) runs on a server with an identical specification to that of the FASTX query processor and the default JVM settings were also changed from the default to that identical to FASTX. Experiments used two datasets. The first was the DBLP document library [4], which is a single XML document containing over 10.5 million elements, 2 million attributes, 6 levels and has a size of 439Mb. The second was the XMARK standard benchmark for testing XML databases [5], automatically generated to 226Mb in size, has over 3 million elements, 0.7 million attributes, and 11 levels. The FASTX index was deployed using an Oracle 10g database, running on a Fedora 7 Linux platform, with a 3.0GHz Pentium IV processor and 1GB of RAM.

The queries in this section are largely taken from previous work [1, 2, 17, 16, 22, 23]. Each query was executed five times and the times recorded in milliseconds were then averaged and displayed in table 4.2. The Comparison divides the times of the eXist output by those of FASTX, indicating that a value of 1 represents an equal score; any figure less than 1 represents a slower run for our approach; and figures greater than 1 represents an improvement using our

approach, these can be seen in table 4.2.

4.2 Experiments

Table 4 contains the sixteen queries used in our experiments, the *axes* and dataset used for each query, the Number of location Steps in each query (*NoS*), the Node Count returned: *NC* and the *Pattern(s)* applied where relevant.

Query 11 returns: *"All proceedings published after any publication by Amit P. Sheth in 1990"* and contains two location steps which have the descendant and following axes respectively. The node test for location step 1 contains the wildcard character (*) which returns all elements which satisfy the filters contained in the predicates (i.e. author='Amit P. Sheth'). This example shows how FASTX compares to eXist for queries where no pattern may be applied as described in section 3.

The other fifteen queries have at least one pattern applied which, provide excellent response times when compared to eXist, these are shown in the table in figure 4.2. The query number is shown in column 1, followed by the query response time for eXist in column 2 and the times for FASTX in column 3. The comparison times as previously described in this section are shown in column 4.

Using an example for each pattern type, we will discuss the query response times.

4.3 Analysis for Query 3

Find all articles from August 1994.

4.3.1 Classification

The first step in the process is to identify the patterns that are applicable to the query.

Input

/dblp//article[./month = 'August'][./year = '1994']

Output

- The FullPath pattern may be applied to location steps 1 & 2.
- The LeafPath pattern may be applied to location steps 1 & 2.

4.3.2 Pattern Selection

The SQL statement resulting from FullPath.SQLMAP() differs greatly from that of LeafPath.SQLMAP() and the choice between the two is crucial for optimal performance. The cause of this variation in response times is attributed to the time it takes to execute the SQL statements across our index.

Experimental analysis shows us that the LeafPath.SQLMAP() performs better than the FullPath.SQLMAP() when the number of elements returned by certain SQL query segments are very high. To illustrate this we must analyse the two SQL statements and consider their processing times. We will then discuss how the necessary data to make this selection possible is gathered.

LeafPath.SQLMAP():

```
(SELECT PARENT FROM BIT WHERE
FullPath = '/dblp/article/month'
AND VALUE = 'August' AND N_TYPE = ELEMENT)
INTERSECT
(SELECT PARENT FROM BIT WHERE
FullPath = '/dblp/article/year'
AND VALUE = '1994' AND N_TYPE = ELEMENT)
```

FullPath.SQLMAP():

```
SELECT PARENT FROM BIT WHERE NAME =
'month'
AND VALUE = 'August' AND N_TYPE = ELEMENT
INTERSECT
SELECT PARENT FROM BIT WHERE NAME = 'year'
AND VALUE = '1994' AND N_TYPE = ELEMENT
AND PARENT IN
(SELECT PRE FROM BIT WHERE FULLPATH =
'/dblp/article')
```

When the cardinality of the set of nodes returned by the FullPath(s) (i.e. */dblp/article*) is large, the SQL engine must compare a massive set of nodes against those identified by the predicates. In the case of the FullPath.SQLMAP() the last part of the SQL statement is:

```
(SELECT PRE FROM BIT WHERE FULLPATH =
'/dblp/article')
```

This select statement returns 373,228 nodes and the "IN" construct in SQL is inefficient for sets of this magnitude. In contrast the LeafPath.SQLMAP() goes directly to the leaf nodes which are filtered further by the *"AND VALUE ="* and *"AND N_TYPE ="* part of each query segment. This reduces the search space to a much smaller set of nodes, thus, the LeafPath.SQLMAP() is much more responsive in this instance. However, when the number of FullPath instances is relatively small, the FullPath.SQLMAP() is the highest performing of the two.

In order to find the number of FullPath instances in advance of the selection process, we exploit the `CARDINALITY` field in the `FULLPATH_TABLE` which forms part of our index. This metadata construct contains the number of instances for each node in the XML repository and is immediately available to the selection process. We discovered a threshold, T=20,000 nodes, for the maximum number of elements for the FullPath pattern. After this threshold value, the processing algorithm selects the LeafPath pattern as it performs better in this situation.

Table 5: Comparative Times

Query	FPP	LP
12	**255**	296
13	**273**	380
14	156	**54**
15	11,112	**528**
16	**234**	357

To illustrate why it is optimal to correctly select between the FullPath and LeafPath Patterns, table 5 shows their comparative times for queries 12-16. The fastest time in each case is shown in bold. Continuing with our example, the output from this step is:

Output

Apply the LeafPath pattern to location steps 1 & 2.

4.3.3 Query Transformation

The LeafPath.SQLMAP() algorithm is executed and the resulting SQL is output.

Table 4: Experiment Details

	Query	Dataset	Axes	NOS	NC	Pattern
1	/dblp/article[year = '1990']	DBLP	ch	2	6,758	FPP or LP
2	/dblp//author[.='John Sieg Jr.'] /ancestor::phdthesis	DBLP	ch, des, anc	3	1	FPP or LP, NP
3	/dblp//article[./month = 'August'] [./year = '1994']	DBLP	ch, des	2	12	FPP or LP
4	/dblp//inproceedings[author='Jim Gray'] [year='1980']	DBLP	ch, des	2	1	FPP or LP
5	//article/title[sub='2']	DBLP	des, ch	2	413	FPP or LP
6	/dblp//year[.='1990'] /parent::book	DBLP	ch, des, par	3	57	PP

7	//year[.='1990']/parent::book[year = '1990']
	[author = 'Ivan Bratko']
		DBLP	des, par	2	1	PP

8	//author[.='Peter Van Roy'] /parent::mastersthesis	DBLP	des, par	2	1	PP
9	//year[.='1990']/parent::book	DBLP	des, par	2	57	PP
10	/dblp/inproceedings [title='Semantic Analysis Patterns.']/author	DBLP	ch	3	2	FP
11	//*[author='Amit P. Sheth'][year='1990'] /following::inproceedings [author='Serge Abiteboul']	DBLP	des, fol	2	2	NP
12	/site/regions//item [quantity = '1'][payment = 'Creditcard']	XMARK	ch, des	3	2413	FPP or LP
13	/site/regions//item[location = 'United States'] [quantity = '1'][payment = 'Creditcard']	XMARK	ch, des	3	1,801	FPP or LP
14	/site/regions/asia/item [quantity = '1'][payment = 'Creditcard']	XMARK	ch, des	4	209	FPP or LP
15	/site/regions/namerica/item [location = 'United States']	XMARK	ch, des	4	14,957	FPP or LP
16	/site/regions//item [location = 'United States'][quantity = '1'] [.//payment = 'Creditcard']/parent::namerica	XMARK	ch, des, par	4	1	FPP or LP, PP

4.3.4 Query Execution

The SQL statement is executed and the result set for this query is:
{9376937, 9379778, 9384816, 9385619, 9387422, 9388266, 9390605, 9394036, 9405497, 9408325, 9413009, 12609594}
These are the PreOrder values for the nodes which form the final result set (an XML document).

4.4 Analysis for Query 10

Find the author of all proceedings named 'Semantic Analysis Patterns.'

4.4.1 Classification

The filter pattern is used when property 1 holds but property 2 does not. In this instance the child 'author' appears after the predicate therefore the filter pattern is applicable.

Input

/dblp/inproceedings[title='Semantic Analysis Patterns.']
/author

Output

- The FullPath pattern may be applied to location steps 1 & 2.
- The LeafPath pattern may be applied to location steps 1 & 2.
- The Filter pattern may be applied to location steps 3 and thus, allows a two pattern combination using one of the patterns above.

4.4.2 Pattern Selection

The cardinality of the FullPath: *"/dblp/inproceedings"* is 607,777, therefore the filter pattern is used in conjunction with the LeafPath pattern.

Output

- Apply the Filter pattern to location step 3 using the LeafPath pattern for steps 1 & 2.

4.4.3 Query Transformation

The Filter.SQLMAP() algorithm is executed and the resulting SQL is output.

Filter.SQLMAP():

```
SELECT PRE FROM BASEINDEX_TABLE WHERE
NAME = 'author'
AND PARENT IN
(SELECT PARENT FROM BASEINDEX_TABLE
WHERE
FullPath = '/dblp/inproceedings/title'
AND VALUE = 'Semantic Analysis Patterns.' AND
N_TYPE = ELEMENT)
```

Query	Exist	FASTX	Comparison
1	5,831	359	16.24
2	32,168	144	223.39
3	703	78	9.01
4	22,024	115	191.51
5	1,490	150	9.93
6	24,609	909	27.01
7	23,100	7,050	3.28
8	28,821	156	184.75
9	24,321	925	26.29
10	13,859	2,059	6.73
11	35,028	41,452	0.84
12	830	255	3.25
13	1,085	273	3.97
14	101	54	1.87
15	453	528	0.85
16	1,122	234	4.79

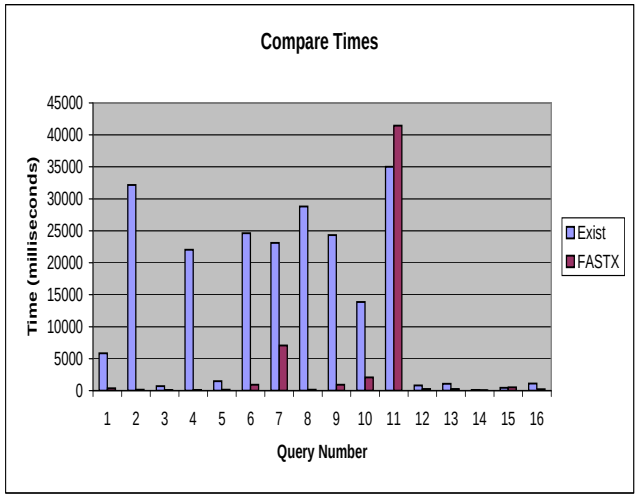

Figure 1: Experiment Times

4.4.4 Query Execution

The SQL statement is executed and the result set for this query is: {1996747, 1996746}

4.5 Analysis for Query 16

Find all payments made in the North American region using a Credit Card when only one item was purchased.

4.5.1 Classification

Example using the Parent pattern.

Input

/site/regions//item[location = 'United States'][quantity = '1'][.//payment = 'Creditcard']/parent::namerica.

Output

- The FullPath pattern may be applied to location steps 1, 2 & 3.
- The LeafPath pattern may be applied to location steps 1, 2 & 3.
- The Parent pattern may be applied to location step 4 combined with one of the patterns above.

4.5.2 Pattern Selection

The number of instances for the FullPath: *"/site/regions-//item"* is 43,500, therefore the Parent pattern is selected in conjunction with the LeafPath pattern.

Output

- Apply the Parent pattern to location steps 1, 2, 3 & 4 using the LeafPath pattern for steps 1, 2 & 3.

4.5.3 Query Transformation

The Parent.SQLMAP() algorithm is executed and the resulting SQL is output.

4.5.4 Query Execution

The SQL statement is executed and the result set for this query is: {663739}

5. RELATED RESEARCH

Many of the techniques employed in the XPath Accelerator [28] have been used in our indexing technique. The XPath Accelerator uses a pre and post order encoding system to store XML data elements in a relational database along with node value and level information. The advantage of this encoding scheme is that it permits traversals from any arbitrary node in the XML document.

We employ a PreOrder encoding scheme based on this method and extend it by adding FullPath information. A FullPath is a metadata construct stored in the index that provides knowledge on data instances in the data tree. Each element in the `BIT` has a FullPath which links it to a `FULLPATH_TABLE` construct. The `FULLPATH_TABLE` provides additional information such as the number of instances of each element and the position of each element in the document tree. We exploit this feature in order to process a group of XPath location steps in parallel, and to correctly select between patterns.

Path summaries are used [1] in a technique which has some similarities to ours. In this instance path summaries (similar to our FullPath) are stored and are used as a means of pruning the search space. An XML dataset may contain elements that appear in structurally different sub-trees. For example, in the `DBLP` [4] dataset elements such as: *title*, *author* and *year* appear as sub-elements of *//inproceedings*, *//book*, and *//article*. For a query such as: *//book[./author='Serge Abiteboul'][./year='1995']* the path summaries: *dblp/book /author* and *dblp/book/year* are used to eliminate from the following sub-trees from the search:

- *dblp/inproceedings/author*
- *dblp/inproceedings/year*
- *dblp/article/author*
- *dblp/article/year*

We extend this approach by identifying query or query segments where we can use FullPath (path summary) metadata to process a group of location steps in parallel. A system of patterns is used in our FASTX database to derive

the highest performing SQL each time an identified pattern emerges for a particular group of contiguous location steps. In instances where two or more patterns may be applied to the same group of location steps, we use the FullPath metadata in the XML repository to select between them.

Translating XPath queries to SQL is fundamental to our parallel processing strategy and we had to address the large issue of impedance mismatch between the two query languages. In [14] we see examples of how XPath queries could be translated to regular XPath expressions and how these expressions are then rewritten as SQL in the presence of well formed XML files using recursive DTDs. In [15] they rewrite XPath logically before translation to SQL begins. This process removes wild cards and eliminates the ancestor/descendant relationships in order to reduce the size of the XPath language. In both cases a large amount of logical rewriting of XPath was required in order to increase efficiency of the derived SQL statements.

While we use similar techniques to transform the XPath query to SQL, we also target a number of XPath location steps in parallel. We look for pre-defined patterns in the query and transform the target location steps to efficient SQL using not only knowledge of the query, but also knowledge of the data within the XML repository. This hybrid approach to XPath query transformation ensures that we always select the highest performing pattern available. In addition, the system is extensible and new patterns may be added where identified in our ongoing work.

Our query engine required only minimal logical rewriting of XPath to deal with wild cards, yet out performs industry leaders as our experiments in section 4 show. Logical rewriting of XPath has the advantage of making the language smaller and more efficient. In [6] they describe the process of optimising XPath to SQL as having two forms: generating optimal SQL directly from the XPath query, or generating suboptimal SQL queries, then attempt to optimise the resulting SQL. In our approach we use the former, which is the approach advocated by [6]. However, their focus is on how each generated SQL statement has a smaller simpler version which may be exploited to improve performance.

We show some similarities to this in our approach as we use a complex selection process to apply the pattern (reusable solution) which derives the highest performing SQL of the identified possibilities. However, we focus on how the structure of the query can impact the SQL resulting from the translation process, and how XML repository information such as the number of instances of elements within the document can complement this process.

6. CONCLUSIONS

In this paper we introduced our framework for XPath optimisation. Our approach to is to optimise XPath query processing by parsing the query itself, extracting metadata which allows us to select the most efficient processing logic for each query or query segment. For this purpose we have identified a number of patterns each of which allow us to process a group of XPath location steps in parallel when certain criteria are met. Our relational index structure enables a more complex selection process in addition to other optimisation techniques. An important factor in our optimisation strategy is the language mapping between XPath and SQL, for which we presented a number of the algorithms.

In the experiments section we looked at a set of standard XPath queries and shown how the application of patterns as described in 3 can allow structurally different queries to be processed by the logic most suiting to them. We have discussed the response times returned by eXist for each of the queries and compared them to that of FASTX. For each pattern type we analysed a query to further emphasise our approach and discussed reasons for the increased response times in each case.

Our ongoing work is focused on identifying new patterns which can process different combinations of XPath axes in parallel using the techniques described in this paper. Furthermore, we aim to provide a more complete mapping process between XPath and SQL using techniques such as the logical rewriting of XPath, and optimising the generated SQL code in a post transformation process.

In [32] Grust et al. describe how standard *off-the-shelf* RDBMSs may be exploited with the appropriate tree encodings to build highly efficient XPath processors using a pre/level/size encoding system. For each node v in this system, pre(v) is the PreOrder value of node v, level(v) is v's distance from the root node and size(v) is v's number of descendants. Region identification using pre/level/size or pre/post information is not part of our current work as we focused primarily on efficient XPath to SQL translation. However, we believe that further optimisation can be achieved by applying the pattern based approach to various other encoding systems and may include region identification and search space pruning techniques as part of our future work.

7. REFERENCES

[1] Barta A., Consens M. and Mendelzon A. Benefits of Path Summaries in an XML Query Optimizer Supporting Multiple Access Methods. *Proceedings of the 31st VLDB Conference*, Morgan Kaufmann, pp 133-144, 2005.

[2] Boulos J. and Karakashian S. A New Design for a Native XML Storage and Indexing Manager. In *Proceedings of EDBT 2006*, LNCS vol. 3896, Springer, pp. 755-772, 2006.

[3] Balmin A., Ozcan F., Beyer K., Cochrane R., and Pirahesh H. A Framework for using Materialized XPath Views in XML Query Processing. *Proceedings of 30th Conference on Very Large Databases*, pp 60-71, Morgan Kaufmann, 2004.

[4] DBLP Computer Science Bibliography (online). `www.sigmod.org/dblp/db/index.html`, 2008.

[5] XMark - An XML Benchmark Project (online). `www.xml-benchmark.org/`, 2008.

[6] Rajasekar Krishnamurthy,Raghav Kaushik, Jeffrey F. Naughton. Efficient XML-to-SQL query translation: where to add the intelligence? proceedings of the Thirteenth international conference on Very large data bases - Volume 30, Toronto, Canada, Pages: 144 - 155, 2004.

[7] Goldman R. and Widom J. DataGuides: Enabling Query Formulation and Optimisation in Semistructured Databases. *Proceedings of the 23rd VLDB Conference*, Morgan Kaufmann, pp 436-445, 1997.

[8] Kaushik R., Shenoy P., Bohannon P. and Gudes E. Exploiting Local Similarity for Indexing Paths in Graph-Structured Data. Proceedings of ICDE, 2002.

[9] Meier W. eXist: An Open Source Native XML Database. In *Web, Web-Services, and Database Systems*, LNCS Vol. 2593, Springer, pp. 169-183, 2002.

[10] Sedna - Native XML Database. `http://modis.ispras.ru/sedna/`, 2008.

[11] MonetDB - open source XML database. `http://monetdb.cwi.nl/`, 2008.

[12] The World Wide Web Consortium. `http://www.w3.org/`, 2008.

[13] Noonan C., Durrigan C. and Roantree M. Using an Oracle Repository to Accelerate XPath Queries. In 17th International Conference on Database and Expert Systems Applications (DEXA 2006), LNCS vol. 4080, pp. 73-82, Springer, 2006.

[14] Wenfei Fan., Jeffrey Xu Yu., Hongjun Lu., Jianhua Lu. and Rajeev Rastogi. Query Translation from XPath to SQL in the presence of recursive DTDs. Proc. of the 31st International Conference on Very Large Databases, ACM, pp. 337-348, 2005.

[15] Jun Gao1., Dongqing Yang1 and Yunfeng Liu1. X2S: Translating XPath into Efficient SQL Queries. Conceptual Modeling for Advanced Application Domains, PSpringer Berlin / CoMWIM 2004. Web Information Integration, Pages 210-222.

[16] Noonan C. and Roantree M. Optimising XML-based Web Information Systems. In Proceedings on International Workshop on Web Information Systems Modeling (WISM), pp. 803-814, Tapir Academic Press, 2007.

[17] O'Connor M., Bellahsene Z. and Roantree M. An Extended PreOrder Index for Optimising XPath Expressions. *Proceedings of 3rd XML Database Symposium*, LNCS Vol. 3671, Springer, pp 114-128, 2005.

[18] Qun C., Lim A. and Ong K. D(k)-Index: An Adaptive Structural Summary for Graph-Structured Data. *Proceedings of the 29th VLDB Conference*, Morgan Kaufmann, 2003.

[19] Noonan C. Masters Thesis. Pruning XML Trees for XPath Query Optimisation. School of Computing, Dublin City University, 2007.

[20] Suciu D. and Miklau G. University of Washington's XML Repository. at: URL http://www.cs.washington.edu/research/xmldatasets, 2002.

[21] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal. Pattern - Oriented Software Architecture: *A System Of Patterns*, ISBN: 0 471 95869 7.

[22] Weigel F. et al. Content and Structure in Indexing and Ranking XML. In *Proceedings of the Seventh International Workshop on the Web and Databases (WebDB)*, pp. 67-72, 2004.

[23] Zhang N. et al. FIX: Feature-based Indexing Technique for XML Documents. In *Proceedings of the 32nd VLDB Conference*, pp.359-370, 2006.

[24] O'Connor M. F., Roantree M. FASTX Repository Processing Framework. *Technical Report ISG-08-01*, Dublin City University, March 2008.

[25] Megginson D., SAX - The Simple API for XML Version 2.0. Online Resource, 2008. *http://www.saxproject.org/*

[26] Document Object Model (DOM) Level 3.0. World Wide Web Consortium, 2004. *http://www.w3.org/DOM/*

[27] PostgreSQL, An Open Source Relational Database System Release 8.3.0. Online Resource, 2008. *http://www.postgresql.org*

[28] Grust T. Accelerating XPath Location Steps. *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pp.109-120, ACM Press, 2002.

[29] Lu Qin and Jeffrey Xu Yu and Bolin Ding *DASFAA*, pp.850-862, Springer, 2007.

[30] Nicolas Bruno and Nick Koudas and Divesh Srivastava *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pp.310-321, ACM, 2002.

[31] Songting Chen and Hua-Gang Li and Junichi Tatemura and Wang-Pin Hsiung and Divyakant Agrawal and K. Selçuk Candan *Proceedings of the 32nd international conference on Very large data bases*, pp.283-294, VLDB Endowment, 2006.

[32] Grust, T., Rittinger, J., and Teubner, J. 2007. Why off-the-shelf RDBMSs are better at XPath than you might expect. In Proceedings of the 2007 ACM SIGMOD international Conference on Management of Data, Beijing, China, June 2007.